# Redirectory

***Release 1.0.0***

**Jul 31, 2020**

# Contents

**Redirectory** is a tool that manages redirects on a cluster level. Requests that would usually end in a **404 PAGE NOT FOUND** can now redirect to new pages specified with custom rules. It binds itself as the default backend (essential a wild card) of your ingress controller and catches all the request that the cluster can't find an ingress rule for.

**KEY FEATURES**

1. Build to run in Kubernetes.
2. Easily scalable by spawning new workers.
3. Can handle multiple domains and sub-domains in a cluster.
4. Every redirect is represented by a redirect rule. Redirect rules support regex.
5. Regex matching performed by Intel's open source Hyperscan regex engine.
6. Can construct new urls by extracting part of old url. For example get an id from the old url and place it in the new one.
7. UI - Easy to use interface so that your marketing people can use it as well.

**AUTHOR** **Kumina B.V.** (Ivaylo Korakov)

CHAPTER 1

# Install

Install Redirectory and creates all the needed resources for it from scratch.

```
helm install --name=redirectory redirectory/conf/helm
```

For more info on installation take a look at the *Installation*.

## 1.1 Documentation

This part of the documentation will show you how to get started using Redirectory.

### 1.1.1 Overview

#### The problem

A lot of big companies have large websites that are constantly changing and are dynamic. This is really nice in order to keep you brand/site up to date with new trends but it also has a bad side effect. Old web pages get deleted and people opening them are getting 404 errors. Usually companies are familiar with that and they even know which old url should redirect to which new one but unfortunately there isn't an easy way to do that in kubernetes at the moment.

#### The solution

The **Redirectory for Kubernetes** project aims to solve this problem once and for all of the companies. It aims to provide a set of features which makes it easy for people of Kumina or customers of Kumina to manage their redirects on their Kubernetes clusters. The project will live on the ingress level in a cluster and will intercept all requests that the ingress is not able to serve and otherwise would send out a 404. Redirectory will catch those errors and try to find the best new url to redirect to in order for the customer to have a seamless experience even though they might be using old and inactive urls.

### 1.1.2 Usage

This part of the documentation assumes you already have Redirectory setup and running on a Kubernetes cluster and you have access to the User Interface provided by the management pod.

#### Overview

This is a piece of software for redirecting requests that would usually end up with a 404 response to a new destination specified by given rules. It is made to work and take advantage of a Kubernetes environment. What you are currently looking at is the so called "management panel" or whatever you would like to call it.

From here you can manage amd access all of the features provided by Redirectory. This User Guide aims to show you how you can use it! Lets begin with the rules.

#### Rules

Rules are the main things that tells Redirectory how to redirect the incoming requests. This section will show you how to:

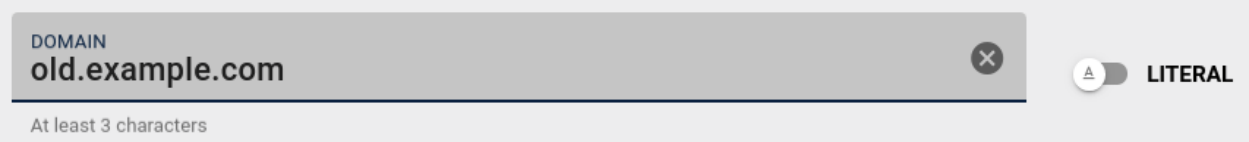
1. Create new rules
2. Exit existing rules
3. And delete not needed once

In order for it to redirect lets say:

```
https://old.example.com/.* -> to -> https://new.example.com/
```

we will first need to enter a rule for this. First you will have to go to the Redirect Rule Explorer section.

There underneath the search filters you will find a button **CREATE NEW REDIRECT RULE**: Once clicked a menu with a few options will appear. The first thing to specify is the domain you would like to redirect from. Keep in mind this domain should be configured that it points to the cluster you are using Redirectory in. After you are done with the domain it should look something like this:




The next thing we need to configure is the path of the domain we just added. Lets to this one the same way as the domain. You might have noticed that we have a **(.*)** in the path of the rule.

This is called **Regex** and it is one of the features of Redirectory, If you have a regex expression you need to toggle to switch between **Regex** and **Literal**

See a little bit more info on Regex in the note below.

---

**Note:** **REGEX** A really simple tutorial.

Regex is quite an expansive topic we don't need much to be able to use it. It is used to select text and in our case URLs. Here are most of the things you will need to get started:

| syntax | meaning |
|---|---|
| . | any character |
| \d | just numbers |
| \w | letters and numbers |
| * | zero or more |
| + | one or more |

Now we can chain them together like this:

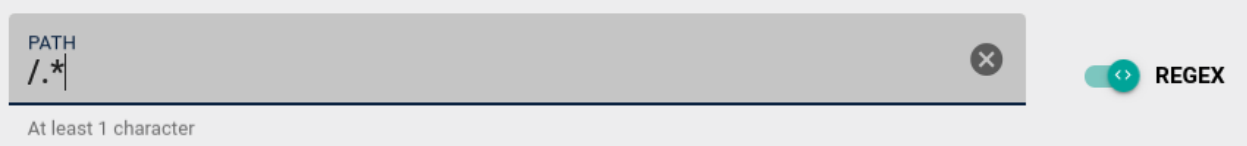
```
/test/path.*
```

which will match any of those:

```
/test/path/any
/test/path/of
/test/path/those
/test/path/123
```

---

Now that we now what we are actually typing in we can fill it in and it should look like the following:




You can fill in the destination the exact same way we did the first two. The last thing that needs to be configured is the weight of a rule. Why do we need it? Sometimes you can get conflicting rules that both of them match the same request. When this happens Redirectory has to know which rules has bigger weight (priority). This is expressed with the weight value of the rule. By default all rules get a weight of 100.

Now we can just create the rule with the **CREATE** button.

### Redirect Rule Explorer

With the Explorer you have all the things you would need in order to manage all of the Redirect Rules for Redirectory. Like we discussed in the Rules section here you can create a new rule but also much more.

On top are the filters. With them you can search through all of the rules you have. You can stack multiple filters to narrow down your search even more. Also keep in mind that for the domain, path and destination filters you can use (*) which is an fnmatch.

---

**Note:** FNMATCH or also called Function Match is a way simpler form of regex. Basically you can have a **(*)** which is equivalent to **(.+)** in Regex and and will match one or more.

---

After you set the filters just press the button **APPLY FILTERS**.

Once you have located the rule that you want in order to view it, edit or delete it you can just click on it: Then the following options will be given for that rule:


Keep in mind the rules are not updated automatically in the User Interface. To make sure your are seeing the latest changes to the rules please click the **REFRESH PAGES** button.

### Bulk Import

But what if I have a lot of rules? For this situation you can make use of the bulk import feature. With it you can upload a CSV (Coma Separated Values) file and all of the rules will be added at once. Because CSV is a basic format a lot of programs support an export to it. You will have to refer to the documentation of the program you are using for more information on exporting the data as CSV.

Take a look at the Bulk Import Section for more information on how the CSV file should be formated in order to get the smooth import.

Once you have uploaded the file the import will begin immediately. The time it takes to process and add all the rules varies on how of course how many you have.

### Ambiguous requests

Ambiguous requests are requests for which Redirectory was unable to decide 100% of what should be the final destination. What does this mean? The main reason of you seeing ambiguous requests is that you have some rules that are not configured correctly.

Sometimes it happens that two or more rules intersect each other and Regex has trouble choosing which one is the more important one because all of them match. Example of intersection:

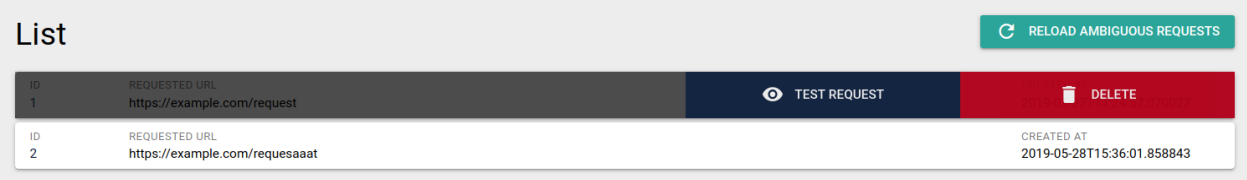
```
1. ggg.test.kumina.nl/test/path/.*
2. \\w+.test.kumina.nl/test/path/.*
3. .*.test.kumina.nl/test/pa.*
```

Now if we make a requests that looks like this:

```
ggg.test.kumina.nl/test/path/aaabb
```

we will match all of the three rules and Redirectory will not know which one should it choose. When this happens Redirectory will always choose the first rule (with the smallest id) and it will also save the request as ambiguous in order for a person to take a look and change the weights of the rules in order not to happen again.

You will be able to see the ambiguous requests section. There are a few options you can make use of in this section. On the top right there is the **RELOAD AMBIGUOUS REQUESTS** button: Once you click an entry/request you are presented with two options. Test option will put this request in the Test Section and show you what is happening behind the scenes. From there you can specify the correct weights for the rules in order to avoid any ambiguous requests in the future. Once you have fixed the issue for a given ambiguous request you can delete it with the second option. See image below for better understanding.




### Hyperscan Database

You have probably noticed that when adding, updating and deleting a rule you have a message that say that the changes will not apply until you compile a new Hyperscan database. This is due to the backend and how Hyperscan works. First make all the changes you would like and then once you are done with all of them you can compile/create a new Hyperscan database.

The settings are located in the Hyperscan Database and Workers Section. Now that you have made the changes you wanted to the rules you can press the **COMPILE NEW HS DB** button. This will create a new Hyperscan Database and apply it to all of the workers. That is everything you need to be worried about with Hyperscan. If you are interested in the workers and how they work please take a look at the next section in the User Guide.

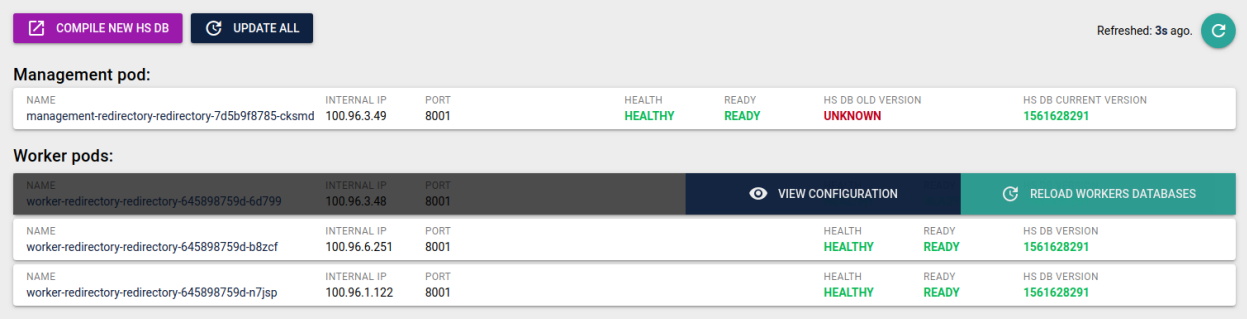
#### Workers and Kubernetes

Redirectory is an application that runs in Kubernetes and makes use of it's scaling features. That is why the application is split into two parts: **management** and **workers**.

The workers are the one that process all of the incoming requests. That is why they need to be up to date with the newest version of the Hyperscan Database. In other words the Redirect Rules.

You will find all of the options for the management and workers in the Hyperscan Database and Workers Section. From there you can see the status of each worker and the current database they have loaded on them. This information updates automatically every 10 seconds or you can click the **REFRESH** button to update now.

The **COMPILE NEW HS DB** button creates a new database and **updates all the workers** after that. If for some reason a worker is out of date you can use the **UPDATE ALL** button or by clicking on the out of date workers and updating it individually. From there you can view the configuration of the workers as well. Take a look at the picture below:




### 1.1.3 Screencast

Take a look at the screencasts to gain a better understanding on how to use the UI to manage all of Redirectory. All the procedures are listed below in no particular order.

#### Bulk Import

#### View Rule

#### Add Rule

#### Edit Rule

#### Delete Rule

#### Search Filters Usage

#### Test Rule

#### Hyperscan DB and Workers

### Navigation

## 1.1.4 Rewrites

Redirectory rules have the ability to be a so called rewrite rule.

A rewrite rule is a rule which can extract a given string from the old url and replace it in the new one and do the redirect. It looks like this:

The user makes a request to:

```
https://asd.test.kumina.nl/id/ac21ca
```

and the new destination should look like this:

```
https://shop.test.kumina.nl/product/id/ac21ca
```

In this case we need to transfer the Id (which stays the same) from the old URL to the new ome. This is done with rewrite rules.

### Explanation

Rewrite rules currently allow you to extract information only from the `path` of the incoming request. You can place the extracted information anywhere you would like in the destination `string`.

The extraction from the path is done with Regex capturing groups. If you don't know them don't worry, they are really simple. Here is an example of a Regex pattern that has a capturing group in it:

```
/test/path/id/(?P<name_of_group>.*)
```

Now if we run the following string (in our case URL):

```
/test/path/id/aa_this_is_in_the_group
```

through the pattern we get the following:

```
{ "name_of_group": "aa_this_is_in_the_group" }
```

Now that we know how to extract values from the path with Regex capturing groups we need to place those values in the destination url and then redirect the user to it. This is done with so called placeholders in the destination url. They look like this:

```
https://www.some.new.website.com/new/shop/{name_of_group}
```

After replacing the values in the placeholder we get this:

```
https://www.some.new.website.com/new/shop/aa_this_is_in_the_group
```

### Examples

Here are a couple of examples for you:

| | **rule** | **regex/rewrite** |
|---|---|---|
| **domain** | test.test.kumina.nl | false |
| **path** | /search/(?P<query>.*) | true |
| **destination** | https://google.com/search?&q={query} | true |

Now you can search in Google through Kumina :)

You can also have multiple values to extract and replace:

| | rule | regex/rewrite |
|---|---|---|
| **domain** | test.test.kumina.nl | false |
| **path** | /shop/(?P<shop_id>[^/]+)/id/(?P<product_id>.*) | true |
| **destination** | https://shop.kumina.nl/{shop_id}/{product_id} | true |

### 1.1.5 Defaults

You added all you rules but you would like to have a default one. If there is no other rule that matches the current request then the default rule will be matched.

Default rules are nothing special. They are just like any other rule you have been adding so far. It is just a wild card rule.

#### Global

Here is an example of a rule that doesn't care about the domain and the path. We can call this rule a **global** default.

| | rule | regex/rewrite |
|---|---|---|
| **domain** | .* | true |
| **path** | .* | true |
| **destination** | https://yahoo.com | false |
| **weight** | 1 | — |

**Tip:** The important thing here is the **weight** of the rule. Default rules must have the lowest possible weight. In our case is 1.

#### Per Domain

The nice thing of having it as a normal rule is that we can make defaults per domain. The only difference is that we need to specify the domain :)

| | rule | regex/rewrite |
|---|---|---|
| **domain** | kumina.nl | false |
| **path** | .* | true |
| **destination** | https://yahoo.com | false |
| **weight** | 2 | — |

**Tip:** It is a good practice to have the domain default rules with **weight** one above the global default rule you have. In our case the global default rule has a weight of **1** therefore this rule should be with **weight** of **2**.

### 1.1.6 Installation

The application is made to run on a **Kubernetes** cluster. There are a few things you need to have in order to deploy it.

1. **Persistent Volume** - In order for the management pod to store the rules (data) in case of a failure or restart. Workers don't have persistent volumes. They sync their data from the management pod.
2. **Role bindings** - Needed because the application must know of worker and management pods. The following permissions are needed for a Role resource:

| resources | verbs |
|---|---|
| endpoints | get, list, watch |
| pods | get, list, watch |

You would be able to find all the **.yaml** configuration files in the Redirectory repository.

#### Installation manually

This installation method is NOT recommended! All of the needed configuration files are located under the folder:

```
$ redirectory/conf/kubernetes
```

You will have to apply all the files manually to your cluster with the following command:

```
$ kubectl apply -f management_ingress.yaml
$ kubectl apply -f management_svc.yaml -f worker_svc.yaml
... and so on
```

You may or may not need to edit the configuration files to fit your particular setup.

#### Installation with HELM

To make the installation easier we are making use of HELM. It is a soft of package manager for Kubernetes but more like a templating engine for Kubernetes **.yaml** configuration files.

If you are not familiar with HELM please take a look at theirs documentation on how to use it: HELM docs

> **Warning:** Before continuing make sure you have HELM installed on your kubernetes cluster. Also make sure you have the Docker images available

#### Install

Install Redirectory and creates all the needed resources for it from scratch.

```
$ helm install --name=redirectory redirectory/conf/helm
```

#### Update

Updates only the resources/things that have changes since the last update or install of Redirectory

```
$ helm upgrade redirectory redirectory/conf/helm
```

#### Delete

Deletes Redirectory from the Kubernetes cluster.

> **Warning:** When deleting the application like this it will also DELETE ALL it's data. You will not be able to get the data back.

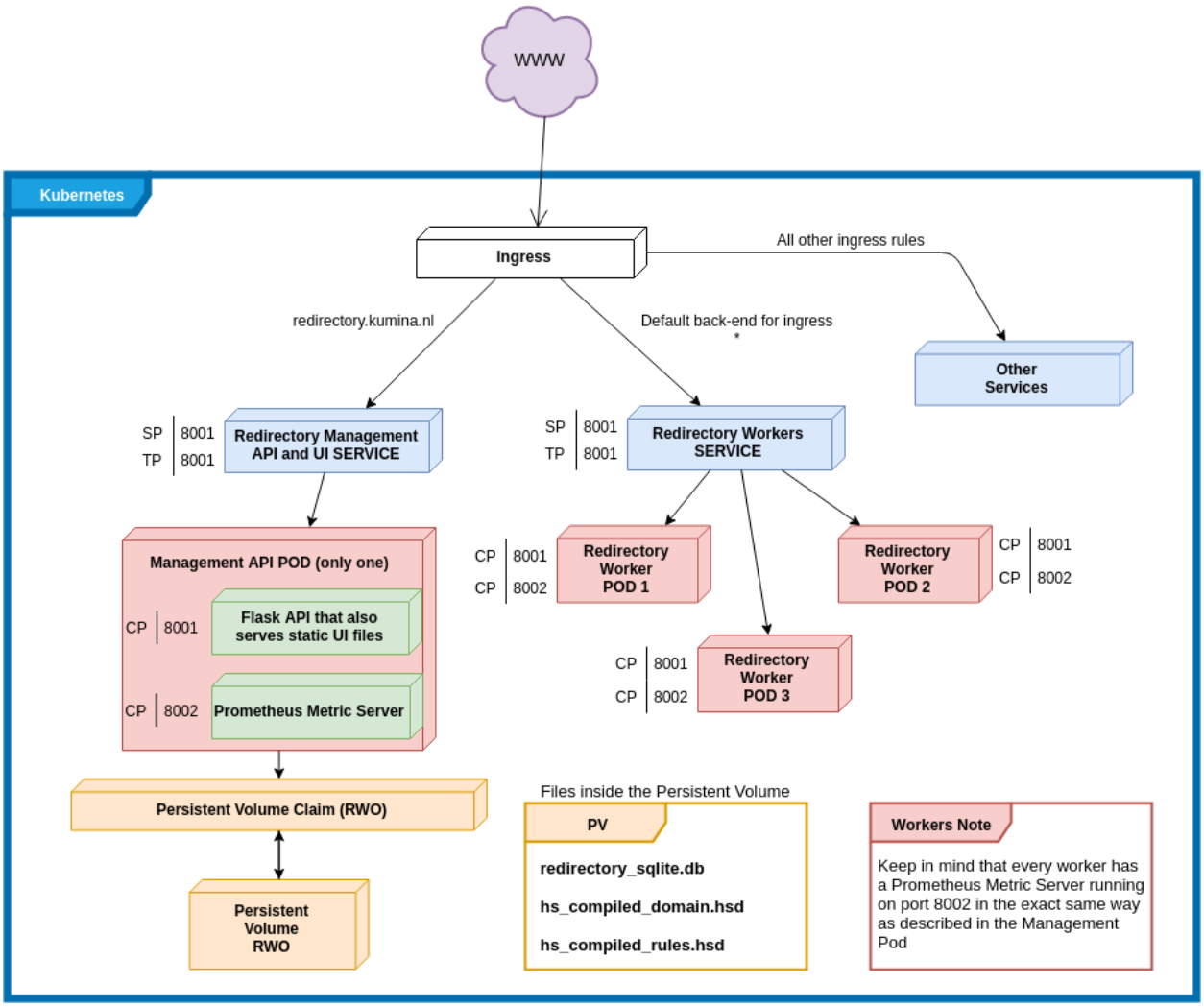
```
$ helm delete --purge redirectory
```

### 1.1.7 Kubernetes

Redirectory is meant to run in a Kubernetes cluster. Kubernetes is a really huge topic and it will not be covered in this documentation. Let's call it a pre-requisite. If you would like to get started you can check the official get started guide.

The application is split into two parts. The worker pods which only handle redirecting requests and a management pod which handles all other functionalities of the application.

To gain a better understanding of how the application runs in Kubernetes please refer to the diagram below.




### 1.1.8 Testing

For the Redirectory project unit testing is encouraged! The library of choice to help us with implementing the unit tests is called **PyTest** and you can see their docs at: pytest docs.

#### Set up

Before we start testing Redirectory let's setup our testing environment. There is already a nice `requirements_test.txt` file we can use for this. You can create an environment with the following moment:

```
mkvirtualenv redirectory_test -r requirements_test.txt
```

#### Running the tests

We can run the tests with the following command:

```
PYTHONPATH=. pytest
```

and if you would like to see the `stdout` while the tests are running:

```
PYTHONPATH=. pytest -s
```

### Structure

Because we make use of **pytest** the tests folder is split into two as shown bellow:

```
tests
├── cases
│   ├── database
│   └── hyperscan
├── fixtures
│   ├── configuration.py
│   ├── database_ambiguous.py
│   ├── database_empty.py
│   ├── database_populated.py
│   └── hyperscan.py
```

Fixtures are functions that will run before every test. Let's say that a certain test needs an already loaded empty database in order to run. We can create a fixture `database_empty` and add it as a requirement to this particular unit test.

This is how the `database_empty` fixture would look like:

```
@pytest.fixture
def database_empty(configuration):
    # Import DB Manager first before the models
    from redirectory.libs_int.database import DatabaseManager

    # Import the models now so that the DB Manager know about them
    import redirectory.models

    # Delete any previous creations of the Database Manager and tables
    DatabaseManager().reload()
    DatabaseManager().delete_db_tables()

    # Create all tables based on the just imported modules
    DatabaseManager().create_db_tables()
```

**Tip:** Fixtures can be added as requirements for other fixtures. In this case before we can init the database we need to make sure the `configuration` is available.

and the unit test will look like this:

```
def test_add_ambiguous_request(self, database_empty):
    """
    Test Description ...
    """
    # Get session
    from redirectory.libs_int.database import DatabaseManager
    db_session = DatabaseManager().get_session()
```

(continues on next page)

(continued from previous page)

```
    # Here is your actual test
    assert True

    # Return session
    DatabaseManager().return_session(db_session)
```

**Must do**

Always return the session to the database before your the end of your test

### 1.1.9 Documentation

The documentation is done with Sphinx. This page will show you how to build the documentation in case you would like to add something to it.

#### Preparation

We need an environment with the specified packages for the documentation. We can create a new env like this:

```
$ mkvirtualenv redirectory_docs -r requirements_docs.txt
```

Now that we have an env we can build the docs but first need to specify one environment variable that points to the folder which holds the `config.yaml` file. Here is the command for this:

```
$ export REDIRECTORY_CONFIG_DIR=../redirectory/conf
```

#### Build

Make sure you have the right environment and the correct env var for the config file. There is a nice script that will help you with building the docs.

```
$ ./build_docs.sh
```

### 1.1.10 License

Redirectory is released under the BSD 3 Clause.

- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.



## 1.2 API Reference

If you are interested in information about a class, specific function or more this is the place to take a look.

### 1.2.1 Redirectory API Reference

This part of the documentation is for developers who would like to know the insides of the project. Here you will find all of the documentation of the source code of Redirectory.

The project is split into different packages for better structure.
Here is a quick overview of all of the packages that the project consists of:

#### libs_int overview

Libs_int is the main package that holds most of the main logic of the application. The main goal is to move out the logic from the API endpoints themselves and have it in one place. This package holds logic for quite a few things:

1. **Config** - .yaml configuration files
2. **Database** - all the needed classes and methods to interact with the database
3. **Hyperscan** - all of the logic of the Hyperscan regex engine
4. **Importers** - different file importers. At the moment only CSV.
5. **Metrics** - logic about Prometheus metrics
6. **Service** - helper classes and methods for API functionality. Also Gunicorn.

#### models overview

Redirectory uses a **SQLite3** database which sits as a file in the **data folder** of the application. The Models packages contains the different models for the database. Redirectory is using **SQLAlchemy** library to the it's interactions with the database.

### runnables overview

Again because Redirectory is made for Kubernetes we split up the application in three different parts:

1. Management
2. Worker
3. Compiler

Because of this we need a nice way to separate between those different modes. Here the **runnables** come in play. A runnable is a class which makes use of the `run()` method which loads different things and prepares the application to run in the correct mode.

### services overview

The service package is where all of the different API endpoints are situated. Because the application is made for Kubernetes there are a few different modes that Redirectory can run as. Therefore the API endpoints are split in the same manner:

1. Management Endpoints
2. Worker Endpoints

Based on the **node_type** which is specified in the **config.yaml** the different sets of API endpoints are loaded at startup. In other words, if you run the application as **management** you won't be able to call **worker** endpoints and the other way around.

---

**Important:** Keep in mind the **stats** endpoints are loaded in both **management and worker** mode.

---

## Contents

### Libs_Int package

### redirectory.libs_int.config package

### redirectory.libs_int.config.configuration module

**class** redirectory.libs_int.config.configuration.**Configuration**
Bases: object

**path = None**

**values = None**

### redirectory.libs_int.database package

### redirectory.libs_int.database.database_actions module

**class** redirectory.libs_int.database.database_actions.**NoneType**
Bases: object

redirectory.libs_int.database.database_actions.**encode_model**(*model: sqlalchemy.ext.declarative.api.DeclarativeMe parent_class: Any = None, expand: bool = False*) → dict

Encodes a DB instance object of a given model into json

**Parameters**

- **model** – The DB model instance to serialize to json
- **parent_class** – A DB model might inherit from another DB model. Pass the parent class in order to be serialized correctly
- **expand** – to include relationships or not

**Returns** a dictionary with basic data types that are all serializable

redirectory.libs_int.database.database_actions.**encode_query**(*query: list, expand: bool = False*) → list

Loops through all of the objects in a query and encodes every object with the help of encode_model() function. All of the individual encoded models are added into a list and then returned.

**Parameters**

- **query** – the query that you would like to encode
- **expand** – if you should expand relationships in the models

**Returns** a list of dictionaries which are the encoded objects

redirectory.libs_int.database.database_actions.**get_or_create**(*session, model, defaults=None, **kwargs*)

Gets an instance of an object or if it does not exist then create it.

**Parameters**

- **session** – the database session
- **model** – the model / table to ger or create from
- **defaults** – any default parameters for creating
- ****kwargs** – the criteria to get or create

**Returns** a tuple(p,q) p: an instance of the object and q: if it is new or old

redirectory.libs_int.database.database_actions.**get_table_row_count**(*db_session, model_table*) → int

Gets the number of rows in a given database in the given database session

**Parameters**

- **db_session** – the database session to use for db actions
- **model_table** – the model / table

**Returns** integer represent the number of row in the table

redirectory.libs_int.database.database_actions.**sanitize_like_query**(*query_str: str*) → str

Sanitizes a string to be used as a query in the DB. It will replace a * with % only when the star is not escaped.

**Parameters** **query_str** – original string to sanitize

**Returns** sanitized converted string

### redirectory.libs_int.database.database_manager module

**class** redirectory.libs_int.database.database_manager.**DatabaseManager**
Bases: object

**create_db_tables**()
Will create all model's tables associated with the current DatabaseManager base. The creation of those tables is safe. If a table already exists it will not be created again. If the DatabaseManager is not initialized then a ValueError will be raised.

**delete_db_tables**()
Will drop all tables associated with the current base of the DatabaseManager. Every model's table that inherits from this base will be dropped. If the DatabaseManager is not initialized then a ValueError will be raised.

**get_base**()
Gets the current base that all models should inherit from. Once a model inherits from this base it will be associated with it.

**Returns** the current base

**get_session**()
Creates a scoped session with with the help of the session maker. This session is specific to the current thread from where this function is called. If a session already exists it will be returned but if not a new one will be created. If the DatabaseManager is not initialized then a ValueError will be raised.

**Returns** a database session for the current thread

**reload**()

**return_session**(*session*)
Closes the given session and removes it from DatabaseManager to prevent from any further use. Sets the session of the DatabaseManager to None

**Parameters** **session** – the session to remove

redirectory.libs_int.database.database_manager.**get_connection_string**()
Generates a connection string to be passed to SQLAlchemy. The string is created from the current loaded configuration with the help of the Configuration() class. There are two options for both SQLite and MySQL database connections.

**Returns** a connection string for SQLAlchemy to use for an engine

### redirectory.libs_int.database.database_pagination module

**class** redirectory.libs_int.database.database_pagination.**Page**(*items*, *page*, *page_size*, *total*)
Bases: object

redirectory.libs_int.database.database_pagination.**paginate**(*query*, *page: int*, *page_size: int*) → redirectory.libs_int.database.database_pagination.Page
Creates a query with the help of limit() and offset() to represent a page. Also counts the total number of items in the given database.

**Parameters**

- **query** – the query which specifies the model tha paginate

- **page** (*int*) – the page number
- **page_size** (*int*) – how many items per page

**Returns** a Page object with all the items inside

### redirectory.libs_int.database.database_rule_actions module

redirectory.libs_int.database.database_rule_actions.**add_redirect_rule**(*db_session*, *domain: str*, *domain_is_regex: bool*, *path: str*, *path_is_regex: bool*, *destination: str*, *destination_is_rewrite: bool*, *weight: int*, *commit: bool = True*) → Union[redirectory.models.redi int]

Creates a new Redirect Rule from all of the given arguments. If a domain, path or destination is already used it is just going to be re-used in the new rule. Before all that it validates rules which are rewrites to see if they are configured correctly.

Depending on where the check failed different integers will be returned.

**Parameters**

- **db_session** – the database session to use for the DB actions
- **domain** – the domain of the new rule
- **domain_is_regex** – is the domain a regex or not
- **path** – the path of the new rule
- **path_is_regex** – is the path a regex or not
- **destination** – the destination of the new rule
- **destination_is_rewrite** – is the destination a rewrite or not

- **weight** – the weight of the new rule
- **commit** – should the function commit the new rule or just flush for ids

**Returns** Redirect Rule - if all went well 1 (int) - if the check failed for rewrite rule 2 (int) - if the check for already existing rule failed

redirectory.libs_int.database.database_rule_actions.**delete_redirect_rule**(*db_session*, *redirect_rule_id: int*) → bool

Tries to delete a redirect rule with a given id If the rule doesn't exist then false will be returned

**Parameters**

- **db_session** – the database session to use for db actions
- **redirect_rule_id** – the id of the rule to delete

**Returns** true if rule deleted successfully else false if rule not found

redirectory.libs_int.database.database_rule_actions.**get_model_by_id**(*db_session*, *model*, *model_id*)

Queries a specific model / table in a given database session for a row with a given ID

**Parameters**

- **db_session** – the database session to use for db actions
- **model** – the model / table to query
- **model_id** – the id of the given model

**Returns** an instance of the model or None if not found

redirectory.libs_int.database.database_rule_actions.**get_usage_count**(*db_session*, *model*, *model_instance_id*) → int

Creates a query that counts the usage of a given model with model_instance_id in the RedirectRule model / table. After that executes the query and returns the result

**Parameters**

- **db_session** – the database session to use for db actions
- **model** – the model to count the usages for
- **model_instance_id** – the id of the model instance

**Returns** an integer representing how many times a certain model with that id is used

redirectory.libs_int.database.database_rule_actions.**update_redirect_rule**(*db_session*, *redirect_rule_id: int*, *domain: str*, *domain_is_regex: bool*, *path: str*, *path_is_regex: bool*, *destination: str*, *destination_is_rewrite: bool*, *weight: int*) → Union[redirectory.models.RedirectRule, int]

Updates the rule with the given ID and with the given arguments. Finds the rule specified with the redirect_rule_id and updates it's values correspondingly. If everything goes correctly then the new version of the rule returned. If no rule with that ID is found an integer is returned If the new rule fails the rewrite validation an integer is returned

**Parameters**

- **db_session** – the database session to use for db actions
- **redirect_rule_id** – the ID of the rule to update
- **domain** – the new domain of the rule
- **domain_is_regex** – the new status of the domain rule
- **path** – the new path of the rule
- **path_is_regex** – the new status of the path rule
- **destination** – the new destination of the rule
- **destination_is_rewrite** – the new status of the destination rule
- **weight** – the new weight of the rule

**Returns** Redirect Rule - which is the updated version if all went well 1 (int) - rule exists but fails validation check for rewrite rule 2 (int) - rule with this id does not exist

redirectory.libs_int.database.database_rule_actions.**validate_rewrite_rule**(*path: str*, *path_is_regex: bool*, *destination: str*) → bool

Checks if all of the needed variables/placeholders in the destination rule are also appearing in the path when compiled to a regex pattern.

**Parameters**

- **path** – the path rule to check for
- **path_is_regex** – if the path rule is a regex (hint in order to pass this check it always has to be)
- **destination** – the destination rule with placeholders in it

**Returns** True if the rule is valid else False

### redirectory.libs_int.hyperscan package

### redirectory.libs_int.hyperscan.hs_actions module

redirectory.libs_int.hyperscan.hs_actions.**get_expressions_ids_flags**(*db_model: sqlalchemy.ext.declarative.api.De*, *expression_path: str*, *expression_regex_path: str*, *id_path: str*, *combine_expr_with: str = None*) → Tuple[List[bytes], List[int], List[int]]

Gets the expression in the correct format from the database. Depending on the arguments the expression can be combined with another piece of data. The expression will also be regex escaped if it is a literal. If the expression is a regex then a second check will be conducted which checks if the expression matches an empty string. If so a different flag than the default is applied.

**Parameters**

- **db_model** – The model/table of the current database
- **expression_path** – The attribute where the expression can be found in the model

- **expression_regex_path** – The attribute holding the value if an expression is regex or not
- **id_path** – The attribute where the id can be found
- **combine_expr_with** – The attribute of extra piece of data that can be appended before the expression

**Returns** a tuple containing the expressions, the ids and the flags. tuple(expressions, ids, flags)

redirectory.libs_int.hyperscan.hs_actions.**get_hs_db_version**() → Tuple[Optional[str], Optional[str]]

Queries the database for the HsDbVersion table which only has one entry at all times. Return the two numbers which represent the old_version and the current_version of the Hyperscan database.

**Returns** tuple of old_version and new_version of the Hyperscan Database

redirectory.libs_int.hyperscan.hs_actions.**get_timestamp**() → str

Gets the current date and time and converts it to epoch

**Returns** an epoch string

redirectory.libs_int.hyperscan.hs_actions.**multi_getattr**(*obj*, *attr*, *default=None*)

Get a named attribute from an object; multi_getattr(x, 'a.b.c.d') is equivalent to x.a.b.c.d. When a default argument is given, it is returned when any attribute in the chain doesn't exist; without it, an exception is raised when a missing attribute is encountered.

redirectory.libs_int.hyperscan.hs_actions.**update_hs_db_version**(*new_db_version: str = None*) → str

Updates the SQLite3 database about the new version of Hyperscan database.

**Returns** the new version of the hyperscan database

### redirectory.libs_int.hyperscan.hs_database module

**class** redirectory.libs_int.hyperscan.hs_database.**HsDatabase**

Bases: object

**static compile_db_in_memory**(*expressions: List[bytes], ids: List[int], flags: List[int]*) → hyperscan.Database

**compile_domain_db**(*expressions: List[bytes], ids: List[int], flags: List[int]*)

**compile_rules_db**(*expressions: List[bytes], ids: List[int], flags: List[int]*)

**db_version = None**

**domain_db = None**

**domain_db_path = None**

**is_loaded = False**

**load_database**()

TODO:

**reload_database**()

**rules_db = None**

**rules_db_path = None**

**save_database**()

### redirectory.libs_int.hyperscan.hs_manager module

**class** redirectory.libs_int.hyperscan.hs_manager.**HsManager**
Bases: object

**database = None**

**static get_error_code**(*error: hyperscan.error*) → int
Hyperscan errors are differentiated by their message instead of an Exception object. This method extracts the error code of a Hyperscan error from the message of that error.

**Parameters** **error** – a Hyperscan error object

**Returns** integer representing the Hyperscan error

**static pick_result**(*db_session*, *redirect_rule_ids: list*) → Tuple[Optional[redirectory.models.redirect_rule.RedirectRule], Optional[bool]]
Checks which of the redirect rules has the largest weight. Gets every redirect rule from the DB and compares their weights. If all the redirect rules have the same weight then the request is considered ambiguous

**Parameters**

- **db_session** – the database session to be used with all DB actions
- **redirect_rule_ids** – a list of all the redirect rule ids

**Returns** the picked redirect rule and if the choice is ambiguous or not

**search**(*domain: str*, *path: str*, *is_test: bool = False*) → Union[list, dict, None]
Searches the two Hyperscan databases for the best match. First it searches the domains to find the right one. Then it combines the id of the domain with the path into a rule. The rule is searched again with the Rule Hyperscan database.

**Parameters**

- **domain** – the domain to search for
- **path** – the path to the domain to search for
- **is_test** – if set to true the function returns the two search context objects for the domain and rule

**Returns** if no match is found int: the id of the redirect rule dict: a dictionary with both the domain and rule search context objects for testing

**Return type** None

**search_domain**(*domain: str*, *domain_search_ctx: redirectory.libs_int.hyperscan.search_context.SearchContext = None*) → Optional[redirectory.libs_int.hyperscan.search_context.SearchContext]
Searches a domain in the hyperscan domain database. Creates a SearchContext object and runs a scan for the domain. Also handles a cancellation of the search which is a hyperscan error with error code -3. If the search doesn't find any matches a None is returned. If there are matches then a SearchContext object will be returned.

**Parameters**

- **domain** – the domain to search for
- **domain_search_ctx** – SearchContext to be passed to Hyperscan

**Returns** None or a SearchContext object

**search_rule**(*rule: str*, *rule_search_ctx: redirectory.libs_int.hyperscan.search_context.SearchContext = None*) → Optional[redirectory.libs_int.hyperscan.search_context.SearchContext]

Searches a rule in the hyperscan rule database. Really similar to the search_domain() method. If the search doesn't find any matches a None is returned. If there are matches then a SearchContext object will be returned.

**Parameters**

- **rule** – the rule to search for. {domain_id}/{path}
- **rule_search_ctx** – SearchContext to be passed to Hyperscan

**Returns** None or SearchContext object

### redirectory.libs_int.hyperscan.search_context module

**class** redirectory.libs_int.hyperscan.search_context.**SearchContext**(*original: str*, ***kwargs*)

Bases: dict

**handle_match**(*destination_id: int*, *from_index: int*, *to_index: int*)

Handles a hyperscan matched passed from the match_event_handler. If the length of the match matches the length of the original search query it will be added to the matched_ids.

**Parameters**

- **destination_id** – the id of the matched expression from Hyperscan
- **from_index** – from where the match starts
- **to_index** – until where the match ends

**is_empty**()

Checks in any matches have been found associated with this context

**Returns** a boolean representing if any matches are found

**matched_ids = None**

**original = None**

### redirectory.libs_int.importers package

This package/folder contains the different importers Redirectory has.

At the moment only one is available but more may be added in the future if they are requested or people contribute.

Follow the links bellow to see the API of the importer.

### redirectory.libs_int.importers.csv_importer module

### CSV Importer

The CSV Importer takes care of importing CSV files containing Redirect Rules and adding them into the SQL database of the management pod.

The behaviour:

1. If a rule in the CSV already exists it is going to be ignored.
2. If a syntax/parsing error occurs somewhere in the CSV file the whole import is marked as **failed** and all of the changes to the database are roll backed.

**class** redirectory.libs_int.importers.csv_importer.**CSVImporter**(*csv_byte_file_in: werkzeug.datastructures.FileStorage*)

Bases: object

A new **CSVImporter** is created for every import and the data of the CSV file is passed as a parameter in the constructor of the class.

**csv_reader = None**
Reader object used to parse the CSV file

**data_template = {'destination': None, 'destination_is_rewrite': None, 'domain': Non**
This is the template that the CSV is checked against. Every row of the CSV must match this template otherwise the whole import will fail

**import_into_db**()
Imports all the rules in the given csv file into the database as RedirectRules. If a rule is a duplicate it will be skipped. If there is an error in parsing the csv then all the changes will be roll backed and the whole import will be marked as fail.

### redirectory.libs_int.metrics package

### Metrics module

Here are a the metrics that are currently being logged by the application:

| name | Description | label names |
|---|---|---|
| redirectory_requests_duration_seconds | Time spent processing requests | node_type |
| redirectory_requests_total | Number of requests processed | node_type, code |
| redirectory_requests_redirected_duration_seconds | Time spend processing a redirect request by label | node_type, measure |
| redirectory_requests_redirected_total | Number of requests that when processed were redirects by label | node_type, code, request_type |
| redirectory_hyperscan_db_compiled_total | Number of times the management pod has compiled the hyperscan db | node_type |
| redirectory_hyperscan_db_reloaded_total | Number of times the worker pod has reloaded the hyperscan db | node_type |
| redirectory_hyperscan_db_version | The version of the hyperscan database by node_type | node_type |

Please fill free to request more that are not in here but you thing might be useful. You can fill in a github issue.

redirectory.libs_int.metrics.metrics.**start_metrics_server**()
Starts a http server on a port specified in the configuration file and exposes Prometheus metrics on it. Also removes GC_COLLECTOR metrics because they are not really needed.

redirectory.libs_int.metrics.metrics.**update_rules_total**()
This function updates the RULES_TOTAL metric every time it is called with a count from the DB

### redirectory.libs_int.service package

### redirectory.libs_int.service.api module

**class** redirectory.libs_int.service.api.**Api**(*app=None*, *version='1.0'*, *title=None*, *description=None*, *terms_url=None*, *license=None*, *license_url=None*, *contact=None*, *contact_url=None*, *contact_email=None*, *authorizations=None*, *security=None*, *doc='/'*, *default_id=<function default_id>*, *default='default'*, *default_label='Default namespace'*, *validate=None*, *tags=None*, *prefix=''*, *ordered=False*, *default_mediatype='application/json'*, *decorators=None*, *catch_all_404s=False*, *serve_challenge_on_401=False*, *format_checker=None*, ***kwargs*)

Bases: flask_restplus.api.Api

**base_path**
The API path

**Return type** str

### redirectory.libs_int.service.api_actions module

redirectory.libs_int.service.api_actions.**api_error**(*message: str, errors: Union[str, list], status_code: int*)

Returns an api error with a given status and a message/messages

**Parameters**

- **message** – A overall message of the error. E.g. Wrong input.
- **errors** – A message in str format or a list of strings for multiple error messages
- **status_code** – The status of the error. E.g. 404, 503 ..

### redirectory.libs_int.service.gunicorn_server module

**class** redirectory.libs_int.service.gunicorn_server.**GunicornServer**(*app*, *options=None*)

Bases: gunicorn.app.base.BaseApplication

This class provides the ability to run gunicorn server from inside of python instead of the running it through the command prompt Gives you a nicer way to handle it and you can override key methods to make it more specific for our use case

**static get_number_of_workers**(*is_worker: bool = False*)
Calculates the number of workers the gunicorn server will use

**init**(*parser*, *opts*, *args*)

**load**()

**load_config**()
This method is used to load the configuration from one or several input(s). Custom Command line, configuration file. You have to override this method in your class.

**static load_metric_server**()
When run() is called on Gunicorn it starts a new process with this flask App. The metric server must run in the same process as the Flask API in order to share the metrics. This function starts the metric server when the Flask APP is loaded into the new process.

### redirectory.libs_int.service.namespace_manager module

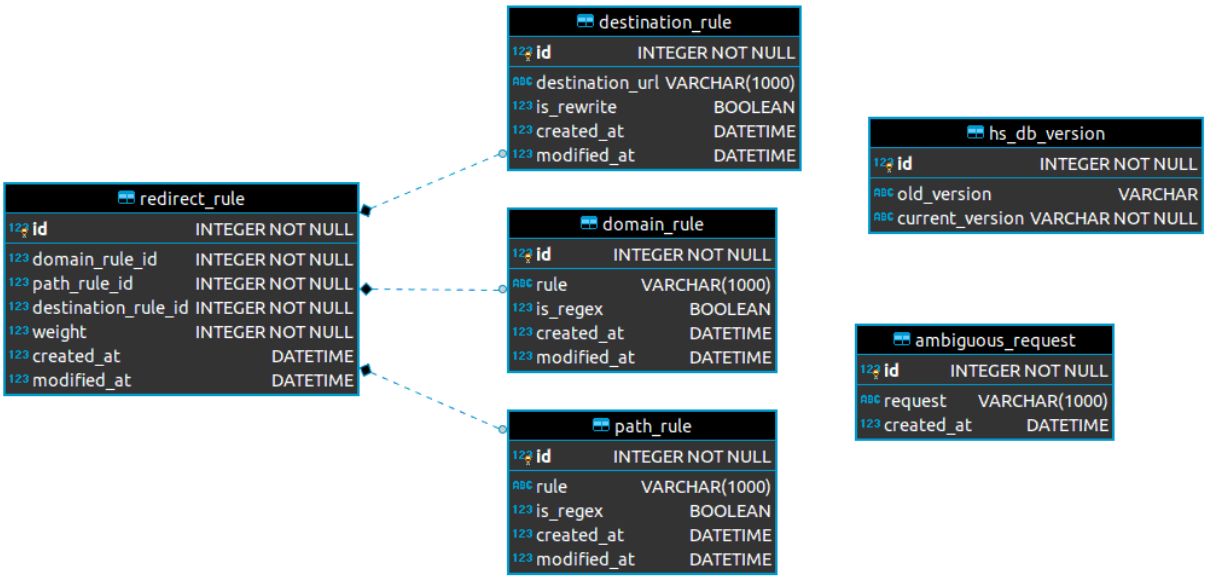
**class** redirectory.libs_int.service.namespace_manager.**NamespaceManager**
Bases: object

**get_namespace**(*name: str*)

**namespace_map = {}**

### Models package

Here as a diagram of the simple database followed by their corresponding classes. I think they are simple enough to understand directly ;)




The `redirect_rule`, `domain_rule` , `path_rule` and `destination_rule` tables all have the following two fields:

| name | Description | type | other |
|---|---|---|---|
| created_at | The time this entry was created on | Datetime | now |
| modified_at | The last time the entry was modified | Datetime | now |

### Redirect Rule

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| domain_rule_id | The ID of the domain rule | Integer | foreign key |
| path_rule_id | The ID of the path rule | Integer | foreign key |
| destination_rule_id | The ID of the destination rule | Integer | foreign key |
| weight | The weight/priority of this rule over the others | Integer | 100 |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    domain_rule_id = Column(Integer, ForeignKey('domain_rule.id'), nullable=False)
    path_rule_id = Column(Integer, ForeignKey("path_rule.id"), nullable=False)
    destination_rule_id = Column(Integer, ForeignKey("destination_rule.id"),␣
↪nullable=False)
    weight = Column(Integer, nullable=False, default=100)
```

### Path Rule

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| rule | The rule that can be regex or literal in a string | String | required, not null |
| is_regex | If the rule is a regex or literal | Boolean | False |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    rule = Column(String(1000))
    is_regex = Column(Boolean, default=False)
```

### Domain Rule

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| rule | The rule that can be regex or literal in a string | String | required, not null |
| is_regex | If the rule is a regex or literal | Boolean | False |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    rule = Column(String(1000))
    is_regex = Column(Boolean, default=False)
```

### Destination Rule

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| destination_url | The destination URL that can have also placeholders | String | required, not null |
| is_rewrite | Weather or not the URL has placeholders in it | Boolean | False |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    destination_url = Column(String(1000))
    is_rewrite = Column(Boolean, default=False)
```

### Ambiguous Requests

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| request | The full URL of the request that the worker got | String | required, not null |
| created_at | The time this entry was created on | Datetime | now |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    request = Column(String(1000), unique=True)
    created_at = Column(DateTime, default=datetime.now())
```

### Hyperscan DB Version

| name | Description | type | other |
|---|---|---|---|
| id | The primary key | Integer | auto increment |
| old_version | The previous version of the HS database | String | nullable |
| current_version | The current loaded version of the HS database | String | required, no null |

This is how it looks in Python:

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    old_version = Column(String, nullable=True)
    current_version = Column(String, nullable=False)
```

### Runnables package

#### redirectory.runnables.compiler module

**class** redirectory.runnables.compiler.**CompilerJob**(*done_callback_function: callable = None*)

Bases: *redirectory.runnables.runnable.Runnable*

**done_callback_function = None**

**run**()

### redirectory.runnables.management module

**class** redirectory.runnables.management.**ManagementService**
Bases: *redirectory.runnables.runnable_service.RunnableService*

**run**()

### redirectory.runnables.runnable module

**class** redirectory.runnables.runnable.**Runnable**
Bases: abc.ABC

**config = None**

**run**()

### redirectory.runnables.runnable_service module

**class** redirectory.runnables.runnable_service.**RunnableService**
Bases: *redirectory.runnables.runnable.Runnable*, abc.ABC

**api = None**

**application = None**

**host = None**

**port = None**

### redirectory.runnables.worker module

**class** redirectory.runnables.worker.**WorkerService**
Bases: *redirectory.runnables.runnable_service.RunnableService*

**run**()

### Services package

This package contains all endpoints that Redirectory has. Just like other parts of the application the API Endpoints are also split into different parts:

1. **Management** - All endpoints for management and UI
2. **Worker** - All endpoints for workers
3. **Status** - Endpoints for watching the status of the application
4. **Root** - Endpoints that are bound to **/** (root path). UI for **management** and redirect for **worker**

### Contents

### Worker Endpoints

#### Worker Get HS DB Version Endpoint

Endpoint: Worker Get Hyperscan DB Version

Method: GET

**RESPONSES:**

- 200: Returns the current Hyperscan DB version that the worker is using
- 400: The worker has not Hyperscan DB loaded at the moment

The Get Hyperscan DB Version endpoint provides with the ability to retrieve the current Hyperscan DB Version that the HsManager() has loaded and is using to run queries. If the worker still has not Hyperscan DB loaded then a 400 is returned.

#### Worker Reload HS DB Endpoint

Endpoint: Worker Reload Hyperscan Database

Method: GET

**RESPONSES:**

- 200: A thread has started with the task of reloading the Hyperscan database

The Reload Hyperscan Database endpoint provides with the ability to start a thread with the task of reloading the Hyperscan Database. The main is the Hyperscan Database but also the SQL database is reloaded as well. When the thread starts it find the management pod with the help of the Kubernetes API and downloads a zip file from it containing all the needed file to reload itself. The zip file is then extracted and first the SQL manager is reloaded and after that the Hyperscan database

### Status Endpoints

#### Status Health Check Endpoint

Endpoint: Status Health

Method: GET

**RESPONSES:**

- 200: Service is up and running

A really simple endpoint that just returns a status OK. Useful for Kubernetes to know if the service has started and it's running. For more in depth check see the Status Readiness. The endpoint returns the same no matter the Node Configuration.

#### Status Readiness Check Endpoint

Endpoint: Status Readiness

Method: GET

**RESPONSES:**

- 200: Service is up and running with a loaded Hyperscan DB
- 400: Service is running but not ready yet. No Hyperscan DB loaded yet

This endpoints acts as a Readiness check for Kubernetes. If the node is of type management then it will always be ready. For management pod the Hyperscan Database doesn't matter. It is used only for testing. If the Hyperscan Database is loaded then the Node can server requests and therefor it is ready. If the Hyperscan Database is NOT loaded yet then the Node is not ready to server requests.

### Status Get Node Configuration Endpoint

Endpoint: Status Get Node Configuration

Method: GET

**RESPONSES:**

- 200: The configuration as JSON is returned

The Status Get Node Configuration provides the ability to see the configuration of the current Node. After the configuration is loaded (which is one of the first things that the application does) it is in dictionary form and is easily serializable to JSON and returned.

### Management Ambiguous Endpoints

### Management Add Ambiguous Request Endpoint

Endpoint: Management Add Ambiguous

Method: POST

**RESPONSES:**

- 200: The ambiguous entry has been added
- 400: An ambiguous entry like this already exists

The Add Ambiguous endpoint provides the ability to add an ambiguous entry to the sqlite database.

### Management Delete Ambiguous Request Endpoint

Endpoint: Management Delete Ambiguous

Method: POST

**RESPONSES:**

- 200: The ambiguous entry has been deleted
- 404: An ambiguous entry with with this id does not exists

The Delete Ambiguous endpoint provides the ability to delete an ambiguous entry from the sqlite database.

### Management List Ambiguous Request Endpoint

Endpoint: Management List Ambiguous

Method: GET

**RESPONSES:**

- 200: A list of all ambiguous request entries
- 404: No ambiguous entries in the SQL database

The List Ambiguous endpoint provides the ability to list all currently stored ambiguous request entries in the SQL database.

### Management Database Endpoints

### Management Compile HS Database Endpoint

Endpoint: Management Compile Hyperscan Database

Method: GET

**RESPONSES:**

- 200: Doesn't matter it will always return a done status
- 400: Unable to compile new hyperscan database

The Compile Hyperscan Database endpoint provides you with the ability to compile a new Hyperscan Database from the current SQLite3 database which holds all the Redirect Rules.

TODO: Make it work with Jobs

### Management Get HS DB Version Endpoint

Endpoint: Management Get Hyperscan DB Version

Method: GET

**RESPONSES:**

- 200: Returns the old_version and the current_version. If not versions are yet available then None

The Get Hyperscan DB Version endpoint provides with the ability to retrieve the previous and the current Hyperscan DB Version which are stored in the database. It will return None for both if there is still no entry about versions in the database.

### Management Reload Management HS DB Endpoint

Endpoint: Management Reload Hyperscan Database

Method: GET

**RESPONSES:**

- 200: The Hyperscan Database has been reloaded

This endpoint provides the management pod with the ability to reload it's hyperscan database that it uses for testing purposes.

### Management Reload Worker HS DB Endpoint

Endpoint: Management Database Reload Worker

Method: POST

**RESPONSES:**

- 200: The specified worker has started updating
- 400: Unable to update the specified worker. See errors

This endpoint provides the management pod with the ability to send an update request to one specific the worker pod. Before sending an update worker request to the worker the endpoint checks if the worker actually exists by making a health status request. If the health status request fails then the worker is considered unreachable and a 400 is returned. If the health status request succeeds then a second reload worker hs db request is send. If the reload worker hs db requests returns 200 then the worker has started updating itself.

### Management Reload Workers HS DB Endpoint

Endpoint: Management Database Reload Workers

Method: GET

**RESPONSES:**

- 200: All workers have been updated
- 400: Unable to update some or all workers. Look at errors

This endpoint provides the management pod with the ability to send an update request to all of the worker pods that are currently running on the cluster.

### Management Kubernetes Endpoint

### Management Get Management Pod Endpoint

Endpoint: Management Kubernetes Get Management

Method: GET

**RESPONSES:**

- 200: Returns information about the management pod
- 400: Unable to get management pod. Not running in a cluster

This endpoint provides the management pod with the ability to get information about itself. This is done with the use of the Kubernetes API.

### Management Get Worker Pods Endpoint

Endpoint: Management Kubernetes Get Workers

Method: GET

**RESPONSES:**

- 200: Returns a list of worker pods information

- 400: Unable to get workers. Not running in a cluster

This endpoint provides the management pod with the ability to get information about all of the worker pods. This is done with the use of the Kubernetes API. It returns an array with every worker as an object. If the application is not running in a Kubernetes environment then a 400 will be returned.

### Management Rules Endpoint

### Management Add Rule Endpoint

Endpoint: Management Add Rule

Method: POST

**RESPONSES:**

- 200: A new rule successfully added to the Redirect Rule database
- 400: Something went wrong during adding the new rule. Check the error which specifies which check it failed

The Add Rule endpoint provides the ability to create/add new rule to the Redirect Rule database. While creating the new Redirect Rule it checks if the rule already exists. If it does a 404 is returned. If the Redirect Rule is new then it will be added to the database and a serialized JSON of the new Redirect Rule instance will be returned.

### Management Delete Rule Endpoint

Endpoint: Management Delete Rule

Method: POST

**RESPONSES:**

- 200: A Redirect Rule with that id has been deleted successfully
- 404: A Redirect Rule with that id does NOT exist

The Delete Rule endpoint provides the ability to delete a RedirectRule from the database. It will not take effect for the Hyperscan database. That must be recompiled. The endpoint takes one argument which is the id of the Redirect Rule. If the rule is found it's delete() method will be executed. The delete is custom and it will delete Domain Rules, Path Rules and Destination Rules if they are not used by any other Redirect Rule. For more insights on the topic take a look at delete_redirect_rule() function. If no Redirect Rule with the given id exists then a 404 will be returned.

### Management Update Rule Endpoint

Endpoint: Management Update Rule

Method: POST

**RESPONSES:**

- 200: The Redirect Rule with that id was successfully updated
- 400: Something went wrong during updating of the Redirect Rule. Check the error message for more info

The Update Rule endpoint provides the ability to update the information for a given Redirect Rule in the database. In the post data all of the needed information for the creation of a rule is specified including the Redirect Rule ID which points to which rule you wish to update.

If a Redirect Rule with that ID is not found then a 400 is returned. If the new Redirect Rule fails the rewrite check 400 will be returned as well.

For more information on how the update rule process works take a look at update_redirect_rule() function.

### Management Get Rule Endpoint

Endpoint: Management Get Rule

Method: POST

**RESPONSES:**

- 200: A Redirect Rule with that ID exists and returned in serialized form
- 404: A Redirect Rule with that id does NOT exist

The Get Rule endpoint provides the ability to retrieve a Redirect Rule by a given ID specified in the post data of the request. If a Redirect Rule with that id doesn't exist then a 404 is returned. If a rule with that id exist then it is serialized and returned.

### Management Get Page Endpoint

Endpoint: Management Get Page

Method: POST

**RESPONSES:**

- 200: A page of redirect rules has been successfully retrieved
- 404: A page with that page number doesn't exists or there are no rule to paginate

The Get Page endpoint provides the ability to split up the RedirectRule database into pages with a given size. From this endpoint you can retrieve a given page by number with a given size. The endpoint also accepts filters (optional) which will be applied and the result of the filtered query will be paginated after that. If no items are found with the specified filters then an api_error with error code 404 will be returned.

### Management Bulk Import Endpoint

Endpoint: Management Bulk Import Rules

Method: POST

**RESPONSES:**

- 200: The import of the CSV file has started successfully
- 400: Wrong file type or format of the CSV. Look at returned error message

The Bulk Import endpoint provides you with the ability to upload a CSV file in a specific format in order to add a lot of Redirect Rules all at once. The importing may take some time which depends on how large is the CSV file. That is why the endpoint makes use of threads. Before we pass the file to the thread we conduct some basic validation at first which includes:

1. Is the file of type CSV
2. Are all the columns specified in the file valid

After this validation has passed successfully then the file is handed over to tbe thread and the import process starts.

Notes:

1. If duplicate Redirect Rules are encountered in the CSV file they will be ignored/skipped.
2. If there is a parsing error somewhere in the file then the whole import process fails and all of the so far added Redirect Rules to the DB are rolled back like nothing happened.
3. At the moment there is no way of telling if an import is finished.

### Management Check Request Endpoint

Endpoint: Management Test Request

Method: POST

**RESPONSES:**

- 200: All the information gathered from the test run of the request
- 400: Hyperscan database not loaded. Can't make search requests

The Test Request endpoint provides the ability to test a request on how it would be process and redirect in a real world scenario. In the post data you specify the request_url which will be ran just as a normal redirect from a worker would. The difference is that not just the final redirect id is returned. A lot of data that might be useful for debugging is exposed as well.

Steps:

1. The request url is parsed and "host" and "path" are extracted from it
2. The Hyperscan Manager is called to search but in test mode
3. After the search is complete we convert all the IDs into their corresponding objects
4. Everything is serialized and returned

For more information on how the search is done take a look at HsManager().search() function.

**Good to test with:**

1. https://iirusa.com/epharmasummitwesta
2. https://example.com/test/path

### Management Sync Endpoints

### Management Sync Download Files Endpoint

Endpoint: Management Sync Download

Method: GET

**RESPONSES:**

- 200: Returns a zip file containing all needed files to perform a sync
- 400: Something went wrong during processing of request. See error message.

This endpoint provides the management pod with the ability for worker pods to download all of the three needed files in one as a zip.

Files in zip:

1. sqlite database
2. hyperscan domain database
3. hyperscan rule database

#### Root Endpoints

#### Management UI Endpoint

Endpoint: Management UI

Method: GET

**RESPONSES:**

- 200: A file with that path exits and it is returned
- 404: A file with that path doesn't exist

The Management UI endpoint serves the static html, css and js files that are the UI itself. The path is the path to the file which the frontend requires. If the path is None then the index.html will be served. The UI static files are located in a folder specified in the Configuration of the node itself. The endpoint serves files only from the specified folder. If a file doesn't exists then a 404 is returned.

#### Worker Redirect Endpoint

Endpoint: Worker Redirect

Method: GET

**RESPONSES:**

- 301: A permanent redirect to the correct location
- 404: Unable to find match for this request

The Worker Redirect endpoint is the CORE endpoint of the application. It parses a request into a host and path. It conducts a search with the help of the HsManager() on the host and path. The search returns a list of matched ids of RedirectRules. If the list is larger than one then we pick the final match with the help of HsManager.pick_result() function. If while picking the final result there are two or more rules with the same weight then the request is considered ambiguous and it is added to the Ambiguous Table. Also if the rewrite is not configured correctly then a 404 page will be returned and the request will be also added to the Ambiguous Table for later checking by a person.

## 1.3 Search documentation

If you are looking for something specific try searching the documentation.

- search

# Python Module Index

## r

# Index

## R

## S



## U



## V



## W